

Modèle 3D d'un objet à partir de photographies: Extraction de silhouette

Martin Janin

15 juin 2017

[La totalité du projet est disponible en ligne](#)

Le travail effectué est celui prévu. La majorité de l'étude a été concentrée sur la détection de contours. La segmentation de l'image obtenue étant effectuée par le procédé simple présenté par Baumgart.

Table des matières

1	Introduction	1
2	Choix d'implémentation	2
2.1	Types et structures	2
2.1.1	Type	2
2.1.2	Structure	2
2.2	Complexité	2
2.2.1	Complexité temporelle	2
2.2.2	Complexité spatiale	2
2.2.3	Coût de copie	2
3	Procédé et Résultats	2
3.1	Conversion dans l'espace LAB	3
3.2	1 ^{er} Filtrage - Domaine fréquentiel - Calcul des composantes de texture	3
3.3	2 ^{ème} Filtrage - Domaine spatial - Détection des bordures	3
3.4	Lissage parabolique	3
3.5	Sommation des composantes	4
3.6	Seuillage	4
3.7	Opérations topologiques	4
3.8	Approximation polygonale	4
4	Conclusion	4
A	code	6
A.1	Etablissement des voisinages	6
A.2	Transformée de Fourier	6
A.3	Filtrage spatial	7
A.4	Lissage parabolique	7
A.5	Opérations Topologiques	8

1 Introduction

Le travail consiste en l'implémentation complète d'un script d'extraction de silhouette en python. Celle-ci a entraîné une réflexion importante sur l'optimisation du programme, de la manière d'implémenter les différentes étapes jusqu'au choix des types et structures utilisés.

2 Choix d'implémentation

La totalité du programme est écrite en python. Il comprend certaines commandes en bash afin de suivre l'évolution de l'utilisation de la mémoire vive.

Les modules utilisés sont :

numpy	→ structure et opérations
system	→ suivie de la mémoire
matplotlib	→ tracé et affichage des images
itertools	→ outils sur les générateurs
time	→ suivi du temps d'exécution de chaque étape

2.1 Types et structures

Le programme prend en entrée une image RGB de taille de l'ordre du mégapixel. Traiter des données de cette taille motive des choix qui ne sont pas habituels dans d'autres domaines de l'informatique.

2.1.1 Type

Le type retenu est le type float32 de numpy, conjugué naturellement au type complex64. C'est le meilleur choix pour plusieurs raisons :

- Les opérations sur les flottants sont plus rapides que sur les entiers. En effet, les processeurs sont d'abord conçus pour effectuer des opérations sur des nombres flottants.
- Les 8 bits alloués à l'exposant du type float32 permettent un nombre maximal de 10^{256} ce qui évite tout dépassement de capacité.
- Il est bien plus aisé de normaliser une image de flottants puisque le type gère de lui-même la perte de précision (contrairement aux entiers)
- Du fait des grandes complexités spatiales, il était impossible d'utiliser des flottants sur 64bits.

2.1.2 Structure

En ce qui concerne la structure adoptée, il s'agit des tableaux numpy, pour des raisons d'efficacité. On est amené à utiliser des tableaux à 5 dimensions, réparties comme suit :

- Les 3^{ème} et 4^{ème} correspondent au plan de l'image : indexer par rapport à ces dimensions correspond à choisir un pixel.
- Les deux premières dimensions correspondent au voisinage de chaque pixel. Chaque pixel contient en effet tout son voisinage afin de pouvoir effectuer des opérations dessus.
- La 5^{ème} dimension est celle des composantes. On manipulerait en effet des images composites, à l'instar d'une image RGB, ce qui permet d'effectuer les mêmes opérations sur toutes les composantes simultanément.

2.2 Complexité

Les choix d'implémentation ont principalement été motivés par l'optimisation de la complexité.

2.2.1 Complexité temporelle

En théorie, la meilleure manière d'effectuer des opérations sur plusieurs millions de pixels est de vectoriser les calculs, c'est-à-dire d'effectuer les calculs simultanément. On utilise pour cela des GPU (Graphical Process Unit) dont les circuits sont conçus pour effectuer simultanément des opérations sur des vecteurs. Cependant, programmer de manière efficace sur GPU nécessite une maîtrise de langages de bas niveau conçus spécialement pour. En pratique, les calculs vectoriels ont donc été faits avec numpy. Bien que numpy utilise uniquement le CPU pour toutes les opérations, quand les calculs sont correctement vectorisés, ils sont effectués par des boucles sous-jacentes codées en C. Les opérations vectorielles de numpy sont significativement plus rapides qu'en python et simule donc bien le comportement de calculs vectorisés.

2.2.2 Complexité spatiale

Vectoriser les calculs nécessite cependant de stocker simultanément les variables intermédiaires de millions de calculs dans la mémoire vive. Il en résulte une grande complexité spatiale. En effet, effectuer un calcul sur le voisinage de chaque pixel demande : $\text{TailleDesDonnees} \times \text{TailleDuVoisinage} \times \text{TailleDeLimage} \times \text{NombreDeComposantes}$ ce qui dans l'implémentation proposée est parfois de l'ordre de : $80\text{Octet} \times 500\text{Mpix} \times 10$ soit 4Gio. Les ordinateurs actuels disposent d'une mémoire de l'ordre de 5Gio. L'optimisation spatiale est donc également importante. Comme il n'y a aucun moyen de gérer explicitement la mémoire en python, la solution mise en oeuvre a consisté à coder les fonctions clés de sorte que le calcul se face en place autant que possible.

2.2.3 Coût de copie

Il faut noter enfin, que lorsqu'il s'agit d'effectuer des opérations sur le voisinage de chaque pixel, il est nécessaire de copier un grand nombre de fois l'image en mémoire (cf. structure). Le coût temporel de ces copies est non négligeable même avec l'optimisation sous-jacente de numpy. Cependant, au moyen d'une manipulation bas niveau des mémoires de travail vectorielles (registres vectoriels) d'un GPU, il serait possible de rendre ces coûts de copie négligeables.

3 Procédé et Résultats

On prend pour exemple l'image ci-dessous. Détaillons les différentes étapes de ce procédé. Les différentes étapes ainsi que leurs résultats sont résumés dans un tableau en fin de document



3.1 Conversion dans l'espace LAB

L'espace LAB est un espace colorimétrique standardisé qui a pour but d'offrir une représentation plus proche de la vision humaine. Elle est construite sur trois composante :

- L : représente l'intensité lumineuse. C'est la moyenne des composantes RGB
- A et B : décrivent la couleur. Ce sont respectivement les différences G - R et G - B

La norme LAB définit ces trois composantes comme les moyennes et différences ci-dessus auxquelles sont appliquées des transformations non linéaires complexes. Ces transformations sont dépendantes de la machine. Cependant l'objectif ici n'est pas de s'approcher de la vision humaine mais de produire une image dont l'analyse produira de meilleurs résultats. Ces transformations ont donc été abandonnées et les composantes LAB ont été calculées par simple différences et moyennes.

La complexité de cette conversion est $\theta(\nu(n^2))$ où n est la largeur de l'image (supposée carrée).

3.2 1^{er} Filtrage - Domaine fréquentiel - Calcul des composantes de texture

En plus de l'intensité et de la couleur, un bon indicateur de l'appartenance d'un pixel à un objet est sa texture. On entend par là l'éventuelle répétition périodique d'un motif. On mesure la valeur de texture d'un pixel relative à un motif en effectuant le produit de convolution du voisinage de ce pixel avec le motif en question, et en sommant la réponse obtenue.

On rappelle la définition du produit de convolution. Soient f et g deux signaux d'extension finie.

$$f * g : x \rightarrow \int_{-\infty}^{+\infty} f(t - x) \cdot g(t) dt$$

Dans sa version discrète, avec u et v deux signaux discrets d'extension finie :

$$u * v : n \rightarrow \sum_{k \in \mathbb{Z}} u(k - n) \cdot v(k)$$

Que l'on peut étendre à des signaux discrets de dimension 2 (des images) :

$$u * v : (n, m) \rightarrow \sum_{(k, l) \in \mathbb{Z}^2} u(k - n, l - m) \cdot v(k, l)$$

On définit alors la réponse d'une image I à un motif M par :

$$\text{texture}_M : (i, j) \rightarrow \sum_{(k, l) \in \mathbb{Z}^2} (V(i, j) * M)(k, l)$$

Où $V((i, j))$ désigne un voisinage du pixel (i, j) dans l'image I.

Afin de calculer plus efficacement ces produits de convolution, on utilise la transformée de fourier discrète. En effet, une propriété de cette opérateur que nous noterons DF est :

$$u * v = DF^{-1}(DF(u) \cdot DF(v))$$

Comme on va réaliser de nombreux produits de convolution, l'idée est de calculer la transformée de fourier discrète du voisinage de chaque pixel et de réaliser ensuite une simple multiplication par la transformée de Fourier du motif. Afin d'accélérer encore les calculs, on utilise l'algorithme de transformée de Fourier rapide. Le code est disponible en annexe. La complexité atteinte de la transformée de Fourier de chaque voisinage de l'image est $\theta(\log_2(t) \cdot \nu(t^2 \cdot n^2))$ où t est la taille des voisinages.

On choisit ensuite de multiplier la transformée de fourier de chaque voisinage par des filtres gaussiens, représentant des motifs de base dans le domaine fréquentiel. Chaque filtrage produit une nouvelle composante de l'image que l'on rajoute à l'image de départ.

3.3 2^{ème} Filtrage - Domaine spatial - Détection des bordures

Il s'agit ensuite de filtrer les composantes de l'images afin d'en extraire les bordures. Le principe est de remplacer la valeur de chaque pixel par une somme pondérée des pixels de son voisinage. J. Canny a montré que le filtre (i.e. les coefficients de la somme) optimal pour différencier les pas d'intensité du reste de l'image est en bonne approximation la dérivée d'une fonction gaussienne. On obtient après filtrage de chacune des composantes dans de multiples directions les bordures correspondant à chacune d'entre elle. Tous les calculs pouvant être correctement vectorisés, la complexité totale de l'opération est : $\theta(t^2 \cdot \nu(n^2 \cdot x))$ où c est le nombre de composantes de l'image.

Code en annexe

3.4 Lissage parabolique

Cependant, les bordures obtenues tendent à avoir une extension spatiale importante et des bordures issues des composantes de texture, on observe des bordures doublées par

rapport à l'image initiale. Ces doubles bordures s'expliquent par l'extension spatiale du filtrage fréquentiel donnant lieux aux composantes de texture. On a alors recourt à un lissage parabolique.

Pour chaque direction de filtrage, on approxime le voisinage de chaque pixel par un cylindre parabolique d'axe cette direction. On remplace alors la valeur du pixel par :

$$\frac{c}{\text{distance au max local}} = \frac{c^+}{\left(\frac{|b|}{2a^+}\right)}$$

Cette opération permet d'affiner les bordures et de supprimer les doubles bordures.

Code en annexe

3.5 Sommation des composantes

On obtient à la suite de ces opérations de nombreuses composantes :

- Pour les composantes de texture `3couleur + 3couleur × 4carttype = 15composante`
- Pour le filtrage `15composante × 2ecarttype = 30composante`

Pour obtenir l'image de bordure finale, on réalise une somme pondérée de toutes ces composantes. L'approche idéale consiste à déterminer les coefficients de sommation par apprentissage supervisé. Dans notre cas, le choix est une conjonction d'empirisme et de valeurs tirées de la littérature.

3.6 Seuillage

L'image est seuillée pour obtenir une image binaire. Le seuil de 0.3 (Pour une image préalablement normalisée à 1) est un choix empirique

3.7 Opérations topologiques

Afin d'améliorer l'image binaire obtenue, on réalise des opérations topologiques simples. On note \mathbf{I} l'ensemble des pixel de valeur 1 dans l'image binaire. On définit alors les opérations suivantes, prenant en paramètre un ensemble \mathbf{U} :

- Dilatation par \mathbf{U} : $\mathcal{D}_{\mathbf{U}}(\mathbf{I}) = \{p \mid \exists q \in \mathbf{I}, p - q \in \mathbf{U}\}$
- Erosion par \mathbf{U} : $\mathcal{E}_{\mathbf{U}}(\mathbf{I}) = \{p \mid \forall v \in \mathbf{U}, p + v \in \mathbf{I}\}$
- Fermeture par \mathbf{U} : $\mathcal{F}_{\mathbf{U}} = \mathcal{D}_{\mathbf{U}} \circ \mathcal{E}_{\mathbf{U}}$

On réalise alors, les opérations suivantes sur l'image binaire : Une érosion par :

0	1	0
1	1	1
0	1	0

Séquentiellement, des fermetures par :

0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
0	0	0	0	0
0	0	0	0	0

Auquel on applique des rotations dans 8 directions différentes.

Code en annexe

3.8 Approximation polygonale

On extrait d'abord le contour extérieur de l'image obtenue en partant du point le plus haut et en se déplaçant de pixel en pixel. On attribue des priorités aux directions de déplacement afin de parcourir effectivement le contour extérieur dans le sens trigonométrique. Par exemple si le sens de déplacement courant est **Bas**, on appliquera comme ordre de priorité : **Gauche - Bas - Droite - Haut**. Une fois ce contour obtenu, on cherche à l'approximer par un polygone de tel sorte que la distance maximale entre le contour et le polygone soit inférieure à un seuil donné. Pour cela, on initialise le polygone par un segment allant du point le plus haut au point le plus bas puis on itère le procédé suivant sur chaque segment :

- On calcule D la distance maximale du segment au contour
- Si $D > \text{seuil}$ on coupe le segment en deux en ajoutant au polygone le point le plus éloigné du segment.

4 Conclusion

Le résultat obtenu, bien qu'imprécis permet ensuite, en combinant les silhouettes obtenues de plusieurs points de vues, de reconstituer le modèle 3D de l'objet. On a vu comment numpy pouvait permettre par la vectorisation (virtuelle), d'effectuer des opérations complexes sur des millions de données. L'optimisation spatiale et temporelle rigoureuse permet de mener à bien des calculs complexes (transformée de Fourier) en un temps raisonnable ($\approx 45s$)


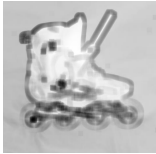





	Opération	Description	Paramètres	Resultats
1	Conversion espace LAB	$(L = \frac{R+G+B}{2})$ code l'intensité, $(A = \frac{G-R+1}{2})$ et $(B = \frac{G-B+1}{2})$ la couleur. Cette espace est plus relevant pour détecter les bordures d'objet.		
2	Calcul des composantes de texture	On réalise la convolution du voisinage de chaque pixel par une gaussienne d'orientation et d'écart type variables. On calcul pour cela la transformée de Fourier de chaque voisinage	$\sigma = 4$ 4 directions	
3	Filtrage	On filtre toutes les composantes par des filtres de Canny (dérivée d'une gaussienne) d'orientation et d'écart type variables.	$\sigma \in \{1, 2\}$, 8 directions	
4	Lissage parabolique	On approxime le voisinage de chaque pixel par une parabole $(ax^2 + bx + c)$ par la méthode des moindres carrés. On remplace l'image par l'image lissée : $I = \frac{c^+}{\text{distance au max local}} = c^+ \cdot \frac{2a^+}{ b }$	8 directions	
5	Sommation des réponses	On somme les 20 composantes obtenues $[(2\text{couleurs} + 4\text{orientation} * 2\text{couleurs}) * 2\text{écart type}]$ en les pondérant par des coefficients piochés dans la littérature.	coefficients de sommations	
6	Seuillage	On seuille l'image pour obtenir une image binaire	seuil = 0.2	
7	Opérations topologiques	On érode (tous les pixels non entourés sont supprimés) puis on dilate (tous les voisins des pixels sont allumés) successivement afin de fermer le contour de l'image	filtres de fermeture	
8	Approximation polygonale	On approxime le contour extérieur de l'objet par un polygone en séparant récursivement les segments d'un polygone jusqu'à atteindre un seuil d'approximation (distance maximale polygone-contour)	seuil d'approximation : 10 pixels	

FIGURE 1 – Tableau Récapitulatif

A code

Dans cette section sont présentées les fonctions principales du code. Elles sont peu commentées, se référer à la documentation numpy pour plus d'information.

A.1 Etablissement des voisinages

On copie le voisinage de chaque pixel. Toujours pour minimiser la complexité spatiale, on utilise des générateurs :

```
def winVect(arr, p, q):
    pr("windows vectorizing", 2)
    a, b = np.shape(arr)[:2]
    la = a - p
    lb = b - q
    makeWin = lambda i, j : arr[i: i + p, j: j + q]
    makeWinGene = lambda l, j : (makeWin(i, j) for i in range(l))
    makeWinColumn = lambda j : np.stack(makeWinGene(la, j), axis = 2)
    winColumnGene = (makeWinColumn(j) for j in range(lb))
    return np.stack(winColumnGene, axis = 3)
```

A.2 Transformée de Fourier

Voici l'implémentation faite de la transformée de Fourier rapide. L'algorithme à été dérécursifié et écrit en place pour assuré une efficacité maximale. Cette fonction peut s'appliquer à un tableau ayant un nombre quelconque de dimension et calculera toujours la transformée de fourier selon les deux premières composantes. Calcul du vecteur d'exponentielle approprié :

```
def exp(n, inverse = False):
    """Generate a vector of exp(2i*pi/n) if inverse is False or \
    exp(-2i*pi/n) otherwise."""
    if inverse:
        eps = 1
    else:
        eps = -1
    expRow = np.exp((eps * 2 * 1j * np.pi / n) * np.arange(n, dtype = ccpl))
    #ccpl is the custom complex type, an alias for np.complex64
    return expRow
```

Calcul de la transformé de fourier selon la deuxième dimension. Cette fonction a été écrite en place pour optimiser la complexité spatiale :

```
def hVectFastFourier2D(arr, inverse = False):
    """Compute the fourier transform of arr on the second dimension \
    IN PLACE."""
    cispow(arr)
    shape = np.shape(arr)
    el = len(shape) - 2
    n = shape[1]
    m = n // 2
    e = exp(n, inverse)
    j = n // 2
    while j >= 1:
        for k in range(j):
            pr(str(j) + " " + str(k), 4)
            temp = arr[:, k: k + j + n: j].copy()
            a = temp[:, : -1: 2]
            b = temp[:, 1:: 2]
            #multDim multiply the two arrays element-wise, on the dimension
            #specified in third argument. It uses numpy broadcasting to optimise
            #the calculus.
            b1 = multDim(b, e[0: n // 2: j], [1])
            b2 = multDim(b, e[n // 2: n: j], [1])
            arr[:, k: k + m: j] = a + b1
            arr[:, k + m: k + n: j] = a + b2
        j = j // 2
    arr[...] = arr / np.sqrt(n)
```

Puis calcul de la transformée de fourier totale :

```
def fastFastFourier2D(arr, inverse = False):
    """Compute the fourier transform of arr on the first \
two dimensions IN PLACE."""
    res = arr.astype(np.complex64)
    hVectFastFourier2D(res, inverse)
    vVectFastFourier2D(res, inverse)
    return res
```

A.3 Filtrage spatial

En pratique, pour implémenter le filtrage spatial, on somme les images décalées et multipliées par les coefficients du filtre :

```
def filt(arr, f):
    """Filter arr with f on his first two dimensions."""
    shape = arr.shape
    a, b = shape[: 2]
    filtShape = f.shape

    #Create a bigger shape to fit the shifted images
    dx = (filtShape[0] - 1) // 2
    dy = (filtShape[1] - 1) // 2
    bigShape = list(shape)
    bigShape[0] += 2 * dx
    bigShape[1] += 2 * dy
    bigShape = tuple(bigShape)

    res = np.zeros(bigShape, dtype = cflt)
    #iteration over the filter
    for i in range(-dx, dx + 1):
        for j in range(-dy, dy + 1):
            #checking if the filter value is non-zero to avoid useless calculus
            if f[dx - i, dy - j] != 0:
                #creating the shifted image
                shiftedArr = np.zeros(bigShape, dtype = cflt)
                shiftedArr[dx + i: dx + i + a, dy + j: dy + j + b] = f[dx - i, dy - j] * cflt(arr)
                #adding it to the result
                res += shiftedArr
    return res[dx : a + dx, dy : b + dy]
```

A.4 Lissage parabolique

La procédure est divisée en 3 sous-fonctions :

- **stackPoint** projète les points de chaque voisinage sur la direction de la localisation. On tire partie des générateur pour assurer une complexité spatiale optimale
- **parabolicApprox** approxime le voisinage de chaque point ainsi projeté en utilisant la méthode des moindres carrés. Les coefficients du système sont calculé dans la dimension la plus faible possible (pas de réplication inutile). On utilise la fonction **solve** de **scipy**, en tirant pleinement partie du broadcasting. On aurait pu recoder le simple pivot de Gauss pour résoudre le système, mais reproduire le broadcasting et la vectorisation de **solve** se serait avéré long et complexe et le temps à manqué.
- **smooth** Calcul à partir du résultat de l'approximation parabolique la fonction lissée.

```
def stackPoints(winArr, u):
    t = winArr.shape[0]
    c = t / 2 - 0.5
    circle = circleCut(t, t / 2).astype(np.bool)
    count = np.sum(circle)
    #Product returns the cartesian product of two generators, usefull to avoid nested loops.
    ind1 = itertools.product(range(t), repeat = 2)
    ind2 = itertools.product(range(t), repeat = 2)
    xGene = (scal((i - c, j - c), u) for i, j in ind1 if circle[i, j])
    yGene = (winArr[i, j] for i, j in ind2 if circle[i, j])
    #The vector of coordinates in the image plane.
    xArr = np.fromiter(xGene, dtype = cflt)
    #The vector of intensity of the pixels.
```

```
yArr = stackGene(yGene, count, winArr.ndim - 2)
return (xArr, yArr)
```

```
def parabolicApprox(xArr, yArr):
    x4sum = np.sum(xArr ** 4)
    x3sum = np.sum(xArr ** 3)
    x2sum = np.sum(xArr ** 2)
    x1sum = np.sum(xArr)
    x0sum = xArr.shape[0]
    yx2sum = np.sum(yArr * xArr ** 2, axis = -1)
    yx1sum = np.sum(yArr * xArr, axis = -1)
    yx0sum = np.sum(yArr, axis = -1)
    mat = np.array([[x4sum, x3sum, x2sum],
                    [x3sum, x2sum, x1sum],
                    [x2sum, x1sum, x0sum]])
    vect = np.stack([yx2sum, yx1sum, yx0sum], axis = -2)
    return np.linalg.solve(mat, vect)
```

```
def smooth(a, b, c, eps):
    aminus = np.maximum(-a, 0)
    cplus = np.maximum(c, 0)
    return (2 * aminus * cplus) / (np.abs(b) + eps)
```

A.5 Opérations Topologiques

Les opérations topologiques ont été implémentées en utilisant astucieusement la fonction de filtrage. `cflt` désigne le type complexe personnalisé qui est un alias pour `numpy.float32`. Dilatation :

```
def expand(arr, f):
    return cflt(filt(arr, f) > 0)
```

Erosion :

```
def erode(arr, f):
    s = np.sum(f)
    return cflt(filt(arr, f) == s)
```

Fermeture :

```
def close(arr, f):
    return erode(expand(arr, f), f)
```

[La totalité du code est accessible en ligne](#)